

Chapter 3: Strings, Vectors and Arrays

The `at()` function and subscript operator `[]` of vectors work similar, but the `at()` function checks if the access is out of bounds of its size (not capacity).

```
vector<int> v{1,2,3,4};

v.at(3) = 7;
cout << "position 3: " << v.at(3) << endl;

v[2] = 8;
cout << "position 2: " << v[2] << endl;

v.push_back(5);

cout << "size of vector: " << v.size() << endl;
cout << "capacity of vector: " << v.capacity() << endl;

cout << "position 6: " << v[6] << endl; // bad but allowed
//cout << "position 6: " << v.at(6) << endl; // try this
cout << "position 10: " << v[10] << endl; //bad but allowed
```

3.4 Introducing Iterators

An iterator is a generalized pointer with a mechanism that lets us:

- identify the position and access the elements in a container
- navigate from one element to another

C++ programs tend to use iterators rather than subscripts to access container elements.

Some advantages of iterators include¹

- 1) similarity to a pointer
- 2) convenience to specify position and iterate through the container.
- 3) code reusability when you wish to switch containers (not all have the `[]` operator).

Only a few standard library types, `vector` and `string` being among them, have the subscript operator. The iterator is more general.

¹ Details on iterators:

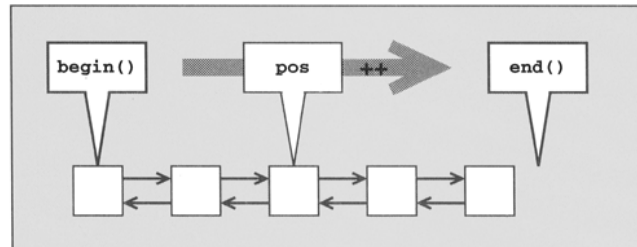
1. URL: <https://www.geeksforgeeks.org/introduction-iterators-c/>

2. URL: <https://www.cprogramming.com/tutorial/stl/iterators.html>

Each container defines its own iterator type.

```
vector<int>::iterator iter;
vector<Sales_item>::iterator it;
set<double>::iterator it2;
```

Each container defines a pair of functions `begin` and `end` that return iterators.



Note that `end()` is an iterator positioned “**one past the end**”

Functions `cbegin` and `cend` return `const_iterator`s. Use iterators if you need to change values, and `const_iterator`s for reading only.

The type of an iterator might seem complicated, but we can use `auto`:

```
vector<int> vec; ...
auto b = vec.begin();
```

The iterator is similar to a pointer: use the dereference operator (`*`) to access the element:

```
*iter = 0;
```

Iterators use the increment operator (`++`) to advance to the next element in the container.

```
++iter;
```

Looping through containers

The subscript operator `[]`

```
// reset all the elements in ivec to 0
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;

for (decltype(ivec.size()) ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

Iterator

```

for (vector<int>::iterator iter = ivec.begin(); iter !=
    ivec.end(); ++iter)
    *iter = 0; // set element to which iter refers to 0

for (auto iter = ivec.begin(); iter != ivec.end(); ++iter)
    *iter = 0;

```

const_iterator for reading but not writing to the elements in the container.

```

string word;
vector<string> text;
while (cin >> word) { // use CTRL+D to end while loop
    text.push_back(word);
}
for (auto iter = text.cbegin(); iter != text.cend(); ++iter)
    cout << *iter << endl;

```

Exercise 3.4 In-class Exercise

Fill up the blank below with an iterator implementation.

```

The original elements in the vector container are: 1 1 1 1 1 1 1 1 1 1
The modified elements in the vector container are: 0 1 0 1 0 1 0 1 0 1

```

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> ivec(10, 1);
    cout << "The original elements in the vector container are: "
        << " ";

    //(fill here)

    cout << endl;
    cout << "The modified elements in the vector container are: "
        << " ";

    //(fill here)

    cout << endl;
    return 0;
}

```

Ans:

```

for (auto iter = ivec.cbegin(); iter != ivec.cend(); ++iter)

```

```

    cout << *iter << " ";

    auto ix = 0;
    for (auto iter = ivec.begin(); iter != ivec.end(); ++iter,
        ++ix) {
        if (ix % 2 == 0) *iter = 0;
        cout << *iter << " ";
    }

```

Key Concept: Generic Programming

We use `!=` rather than `<` in our for loops. In C++ STL, **all of the library containers** have iterators that define the `==` and `!=` operators. Most of those iterators do not have the `<` operator. As a result, `!=` is more general and adaptable for container switching.

Iterator Operations

Standard iterators support only a few operations, which are listed below:

| | |
|-----------------------------|---|
| <code>*iter</code> | Returns a reference to the element denoted by the iterator <code>iter</code> . |
| <code>iter->mem</code> | Dereferences <code>iter</code> and fetches the member named <code>mem</code> from the underlying element. Equivalent to <code>(*iter).mem</code> . |
| <code>++iter</code> | Increments <code>iter</code> to refer to the next element in the container. |
| <code>--iter</code> | Decrements <code>iter</code> to refer to the previous element in the container. |
| <code>iter1 == iter2</code> | Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container. |
| <code>iter1 != iter2</code> | |

Remark: We can compare two valid iterators using `==` or `!=`. Iterators are equal (1) if they denote the same element or (2) if they are both off-the-end iterators for the same container. Otherwise, they are unequal. For example, we can simply write a code fragment that will capitalize the first character of a string.

```

string s("some string");
if (s.begin() != s.end()) { // make sure s is not empty
    auto it = s.begin(); // it denotes the first character in s
    *it = toupper(*it); // make that character uppercase
}

```

3.4.2. Iterator Arithmetic

Iterators for `string` and `vector` support additional operations that can move an iterator multiple elements at a time. They also support all the relational operators. These operations, which are often referred to as **iterator arithmetic**, are described below (Table 3.7).

| | |
|---------------------------------------|--|
| <code>iter + n</code> | Adding (subtracting) an integral value <code>n</code> to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container. |
| <code>iter - n</code> | |
| <code>iter1 += n</code> | Compound-assignment for iterator addition and subtraction. Assigns to <code>iter1</code> the value of adding <code>n</code> to, or subtracting <code>n</code> from, <code>iter1</code> . |
| <code>iter1 -= n</code> | |
| <code>iter1 - iter2</code> | Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container. |
| <code>>, >=, <, <=</code> | Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container. |

3.5 Array

An array consists of a type specifier (such as `int`), an identifier (such as `myArray`, `yourArray`), and a dimension. The type specifier indicates what **type** the elements are stored in the array. The dimension specifies how many elements the array will contain.

The **general form** for declaring an array is:

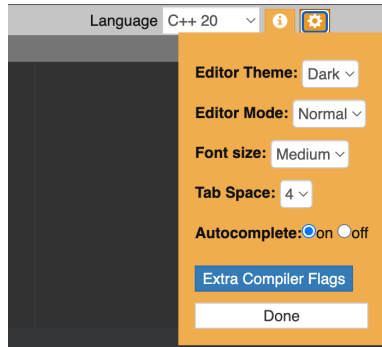
```
| typeName arrayName[arraySize];
```

Unlike the vector, array has fixed size for better run-time performance (but at the cost of having fix length).

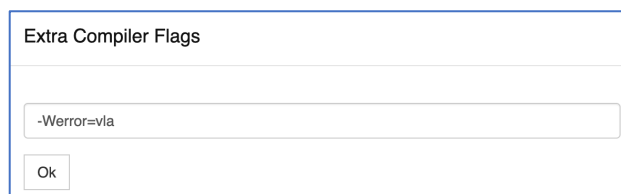
The expression `arraySize`, which is the number of elements, must be a **constant expression** (must be known at compile time).

```
| int arr[10]; // array of ten ints
|
| const unsigned s = 42; // constant expression
| int *parr[s]; // array of 42 pointers to int
```

Variable Length Arrays (VLA) are by default allowed in many compilers, although it is NOT in the C++ standard. To not allow this in onlinegdb.com, set the Extra Compiler Flags:



with “-Werror=vla” without the double quotes



Once you do so, the following code should have an error

```
unsigned cnt = 42; // not a constant expression
string bad[cnt]; // error: cnt is not a constant expression
```

Initialization

If we do not supply explicit initialization, elements in an array are default initialized². In other words, they are left uninitialized.

```
int intArray[3];
string sArray[3];
```

You can do Zero initialization³:

```
int i[3] = {};
```

Or to initialize the array⁴:

² Default Initialization URL: https://en.cppreference.com/w/cpp/language/default_initialization

³ Zero initialization URL: https://en.cppreference.com/w/cpp/language/zero_initialization

⁴ Array initialization URL: https://en.cppreference.com/w/c/language/array_initialization

```
int intArray[3] = {0, 1, 2}; // element initialization
int intArray[] = {0, 1, 2}; // element initialization
char ca1[] = {'C', '+', '+'}; // dim = 3
char ca2[] = {'C', '+', '+', '\\0'}; // dim = 4
char ca3[] = "C++"; // {'C', '+', '+', '\\0'}, dim = 4
```

The end of C++ char arrays is signaled by a special character: the *null character*, whose literal value can be written as '\\0' (backslash, zero)⁵.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s = "abc";
    cout << s.size() << endl; // 3

    char ca[] = "abc";
    cout << sizeof(ca) << endl; // 4

    return 0;
}
```

3.5.2. Accessing the Elements of an Array

As with the library `vector` and `string` types, we can use the subscript operator `[]`.

When we use a variable to subscript an array, we normally should define that variable to have type `size_t`, a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory. The `size_t` type is defined in the `cstdint` header, which is the C++ version of the `stdint.h` header from the C library. You often do not need to explicitly include `cstdint` since many header files have likely included it already.

(Example) suppose we have the following array defined,

```
const size_t array_size = 10;
int ia[array_size];
```

⁵ More details on char array can be found: URL: <http://www.cplusplus.com/doc/tutorial/ntcs/>

(a) How to assign value of each element equal to its index?

```
| for (size_t i=0; i != array_size ; ++i)
|     ia[i] = i;
```

(b) How to copy one array ia into the other ia2?

```
| int ia2[array_size];
| for (size_t i=0; i != array_size ; ++i)
|     ia2[i] = ia[i];
```

We can use range for loop to access the element in an array as we did for string and vector.

```
| #include <iostream>
| using namespace std;
|
| int main(){
|     const size_t array_size = 5;
|     int ia[array_size] = {0, 1, 2, 3, 4};
|     for (auto& i : ia)
|         i += 2;
|     for (auto i : ia)
|         cout << i << " ";
|     cout << endl;
|     return 0;
| }
```

2 3 4 5 6

3.5.3. Pointers and Array

Arrays are pointers, the array variable points to the first element of the array:

```
| int ia[] = {0,2,4,6,8};
| int *ip = ia; // ip points to ia[0]
```

We can use **pointer arithmetic** to compute a pointer to an element by adding (or subtracting) an integral value to (or from) a pointer to another element in the array:

```
| int *ip2 = ip+4; // ip2 points to ia[4]
```

The meaning of the above code is to add from memory address `ip` with value of `4*sizeof(int)`, which results to the memory address of the fifth element in the array `ia`.

More example:

```
int ia[] = {0,2,4,6,8};
cout << *(ia+2); // prints 4
```

Pointers are Iterators

Pointers to array elements support the same operations as iterators on vectors or strings. For example, use the increment operator to move from one element in an array to the next:

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p points to the first element in arr
++p; // p points to arr[1]
```

Library begin and end functions

To make it easier and safer to use pointers, the iterator library includes two functions, named `begin` and `end`.

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
int *beg = begin(ia); // pointer to the first element in ia
int *last = end(ia); // pointer one past the last element in ia
```

We can use the **same syntax** for vector:

A:

```
#include <vector>

...
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
auto beg = begin(v); // or v.begin();
auto last = end(v); // or v.end();
```

Using `begin` and `end`, it is rather easy to write a loop to process the elements in an array.

```
#include <iostream>
#include <iterator>

using namespace std;

int main()
{
    int arr[5] = {1,2,3};
    int *pbeg = begin(arr), *pend = end(arr);
    while (pbeg != pend){
        cout << *pbeg << " ";
        ++pbeg;
    }
}
```

3.2 Library `string` Type

The `string` type supports variable-length character strings. The library takes care of managing the memory and provides various useful operations.

| | |
|----------------------------------|---|
| <code>string s1;</code> | Default constructor; <code>s1</code> is the empty string |
| <code>string s2(s1);</code> | Initialize <code>s2</code> as a copy of <code>s1</code> |
| <code>string s3("value");</code> | Initialize <code>s3</code> as a copy of the string literal |
| <code>string s4(n, 'c');</code> | Initialize <code>s4</code> with <code>n</code> copies of the character <code>'c'</code> |

string input/output (I/O)

The following lines of code read and discard any leading whitespace (e.g., spaces, newlines, tabs), and then read characters until the next whitespace character.

Example:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s;
    cin >> s;
    cout << s << endl;
    return 0;
}
```

Q: what are the outputs if we enter `Hello World!` from inputs?

A:

```
| Hello
```

The following code reads the entire line of string, not including the line break.

Example:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;
    getline(cin, line);
    cout << line << endl;
    return 0;
}
```

Q: what are the outputs if we enter Hello World! from inputs?

A:

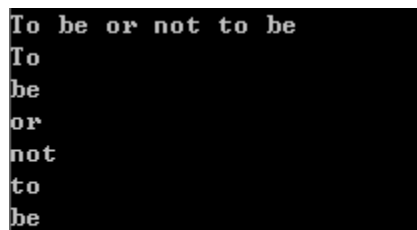
```
| Hello World!
```

The string I/O is often combined with the while loop to read strings:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s;
    while (cin >> s) // CTRL+D to end loop
        cout << s << endl;
    return 0;
}
```

The program read string or strings. Print each string in its own line but discards any whitespace (e.g., spaces, newlines, tabs) before or after the string.



```
To
be
or
not
to
be
```

On Linux/Unix, CTRL+D is the End of File (EOF) signal, and can be issued to stop the program. On a windows computer, it is the CTRL+Z. However, this closes the input stream, and you can no longer use `cin` anymore. A better approach is to use a sentinel value (such as “Q”) to signal the end of inputs.

```
| #include <iostream>
```

```
#include <string>
using namespace std;

int main()
{
    string s;
    do {
        cout << "Please input string: ";
        cin >> s;
        cout << "string: " << s << endl;
    } while (s != "Q");
    return 0;
}
```

Exercise In-class Exercise: Write a program to read lines from input and output the lines with a proper line number. Below is a sample run:

```
TWO roads diverged in a yellow wood,
<Line 1> TWO roads diverged in a yellow wood,
And sorry I could not travel both
<Line 2> And sorry I could not travel both
And be one traveler, long I stood
<Line 3> And be one traveler, long I stood
And looked down one as far as I could
<Line 4> And looked down one as far as I could
```

Answer

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s;
    unsigned lineNo = 0;
    while (getline(cin, s)) // CTRL+D to end loop
        cout << "(Line " << ++lineNo <<") " << s << endl;
    return 0;
}
```

Table 3.2 string Operation

| | |
|---------------------------------------|--|
| <code>os << s</code> | Writes <code>s</code> onto output stream <code>os</code> . Returns <code>os</code> . |
| <code>is >> s</code> | Reads whitespace-separated string from <code>is</code> into <code>s</code> . Returns <code>is</code> . |
| <code>getline(is, s)</code> | Reads a line of input from <code>is</code> into <code>s</code> . Returns <code>is</code> . |
| <code>s.empty()</code> | Returns <code>true</code> if <code>s</code> is empty; otherwise returns <code>false</code> . |
| <code>s.size()</code> | Returns the number of characters in <code>s</code> . |
| <code>s[n]</code> | Returns a reference to the char at position <code>n</code> in <code>s</code> ; positions start at 0. |
| <code>s1 + s2</code> | Returns a string that is the concatenation of <code>s1</code> and <code>s2</code> . |
| <code>s1 = s2</code> | Replaces characters in <code>s1</code> with a copy of <code>s2</code> . |
| <code>s1 == s2</code> | The strings <code>s1</code> and <code>s2</code> are equal if they contain the same characters. |
| <code>s1 != s2</code> | Equality is case-sensitive. |
| <code><, <=, >, >=</code> | Comparisons are case-sensitive and use dictionary ordering. |

The **size** of **string**:

It might be logical to expect that `s.size()` returns an `int` or an `unsigned`. Instead, `s.size()` returns a `string::size_type` value. The reason is that the `string` class—and most other library types—defines several **companion types**. These companion types make it possible to use the library types in a machine independent manner. The type `size_type` is one of these companion types, and is a type for the number of elements in a container.

We use the scope resolution operator (`::`) to indicate that the name `size_type` is defined in the `string` class. In other words, `string::size_type`. **It is an unsigned (integer) type big enough to hold the size of any string.**

It can be tedious to type `string::size_type`. We can ask the compiler to provide the appropriate type by using `auto`

```
string s = "I am a C++ string";
string::size_type len1 = s.size();
auto len2 = s.size(); // len has type string::size_type
```

There are more `string` functions in the `string` class⁶. For example, we can use `find`:

```
#include <iostream>
#include <string>
#include <cstdint>          // std::size_t

int main ()
{
    std::string str ("Please, replace the vowels in this sentence
by asterisks.");

    std::size_t found = str.find_first_of("aeiou");
    // when no match, find_first_of returns std::npos

    while (found != std::string::npos)
    {
        str[found] = '*';
        found = str.find_first_of("aeiou", found+1);
    }
}
```

⁶ More `string` functions and its definitions can be found: URL:

<http://www.cplusplus.com/reference/string/string/>

```

    }

    std::cout << str << '\n';

    return 0;
}

```

Output is:

```

| Pl**s*, r*pl*c* th* v*w*ls *n th*s s*nt*nc* by *st*r*sks.

```

3.2.3 Dealing with Characters in a string

The `ctype.h` from C (included through the `cctype` header in C++) provides useful utility functions to manipulate characters.

For example, we can see how many characters are punctuations:

```

int punct_cnt = 0;
for (string::size_type index = 0; index != s.size(); ++index)
    if (ispunct(s[index])) ++punct_cnt;

for (auto index = 0; index != s.size(); ++index)
    if (ispunct(s[index])) ++punct_cnt;

```

Table 3.3 cctype Functions

| | |
|--------------------------|--|
| <code>isalnum(c)</code> | true if <code>c</code> is a letter or a digit. |
| <code>isalpha(c)</code> | true if <code>c</code> is a letter. |
| <code>isctrl(c)</code> | true if <code>c</code> is a control character. |
| <code>isdigit(c)</code> | true if <code>c</code> is a digit. |
| <code>isgraph(c)</code> | true if <code>c</code> is not a space but is printable. |
| <code>islower(c)</code> | true if <code>c</code> is a lowercase letter. |
| <code>isprint(c)</code> | true if <code>c</code> is a printable character (i.e., a space or a character that has a visible representation). |
| <code>ispunct(c)</code> | true if <code>c</code> is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace). |
| <code>isspace(c)</code> | true if <code>c</code> is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed). |
| <code>isupper(c)</code> | true if <code>c</code> is an uppercase letter. |
| <code>isxdigit(c)</code> | true if <code>c</code> is a hexadecimal digit. |
| <code>tolower(c)</code> | If <code>c</code> is an uppercase letter, returns its lowercase equivalent; otherwise returns <code>c</code> unchanged. |
| <code>toupper(c)</code> | If <code>c</code> is a lowercase letter, returns its uppercase equivalent; otherwise returns <code>c</code> unchanged. |

The following code use the range-for to count the number of punctuations in a string

```
unsigned punct_cnt = 0;
for (auto e: s)
    if (ispunct(e)) ++punct_cnt;
```

Exercise: Write a program to read a line of strings from the standard input and change all the characters to lowercase. A sample run looks like:

```
Enter a line of strings: Hello World
The line in lowercase is: hello world
```

Answer:

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;

int main()
{
    string line;
    cout << "Enter a line of strings: ";
    getline(cin, line);
    for (auto& c : line)
        c = tolower(c);
    cout << "The line in lowercase is: " << line << endl;
    return 0;
}
```

We can use **fstream**

Exercise: A text file `input.txt` contains sentences. A line with an empty string indicates a paragraph break. Write a program to store all the lines in a vector container and print the lines in the first paragraph. For example, if our `input.txt` has the following contents:

```
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim
```

Because it was grassy and wanted wear,
 Though as for that the passing there
 Had worn them really about the same,

And both that morning equally lay
 In leaves no step had trodden black.
 Oh, I kept the first for another day!
 Yet knowing how way leads on to way
 I doubted if I should ever come back.

I shall be telling this with a sigh
 Somewhere ages and ages hence:
 Two roads diverged in a wood, and I,
 I took the one less traveled by,
 And that has made all the difference.

The Road Not Taken by Robert Frost

Answer:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    ifstream fin;
    fin.open("input.txt");
    if (!fin)
    {
        cerr << "cannot open input.txt" << endl;
        return -1;
    }

    string line;
    vector<string> text;
    while (getline(fin, line)) {
        text.push_back(line);
    }

    cout << "The first paragraphs of the input file is" << endl <<
    endl;
    for (auto it = text.cbegin(); it != text.cend() &&
        *it != "\r"; ++it)
        cout << *it << endl;
    return 0;
}
```

Note: ‘\r’ is carriage return; it moves the cursor to the next line. Some system use ‘\n’ and even “\r\n”. This may be complicated. Instead, use `isgraph((*it)[0])`.